NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ATTACKING SOFTWARE CRISIS
A MACRO APPROACH

by

Tahir N. Qureshi

March 1985

Thesis Advisor: Clair A. Peterson

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Attacking Software Crisis A Macro Approach	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis March 1985
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Tahir N. Qureshi	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School Monterey, CA 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School Monterey, CA 93943	12. REPORT DATE: March 1985
	13. NUMBER OF PAGES: 86
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

software, software engineering, software development, software crisis

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This thesis attempts to provide solutions to overcome the software crisis. The basic premise of this thesis is that unless the problems at the software industry level are solved, no number of technical and project management tools can be of much help in overcoming the software crisis. The author examines the existence of the software crisis, its causes and its serious impact on every walk of life. The nature of software development is discussed, considering it as a craft (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ABSTRACT (Continued)

and as an engineering discipline. After evaluating various alternatives, a managerial approach is emphasized. Issues like education, professionalionalization, programmer's productivity, and human factors are discussed. Action on these recommendations requires crossing organizational boundaries, and viewing the problem from a macro perspective.

Attacking Software Crisis
A Macro Approach

by

Tahir N. Qureshi
Lieutenant Commander, Pakistan Navy
M.B.A., University of Karachi, 1983
L.L.B., University of Karachi, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
March 1985

ABSTRACT

This thesis attempts to provide solutions to overcome the software crisis. The basic premise of this thesis is that unless the problems at the software industry level are solved, no number of technical and project management tools can be of much help in overcoming the software crisis. The author examines the existence of the software crisis, its causes and its serious impact on every walk of life. The nature of software development is discussed, considering it as a craft and as an engineering discipline. After evaluating various alternatives, a managerial approach is emphasized. Issues like education, professionalization, programmer's productivity, and human factors are discussed. Action on these recommendations requires crossing organizational boundaries, and viewing the problem from a macro perspective.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. OVERVIEW

> Give a man a hammer, and he will begin to see the world as a collection of nails.

This is precisely what is happening to the software of today. On a random survey of 50 articles from software journals, it was revealed that the authors of 45 of them were mathematicians, physicists, scientists or electrical/electronics engineers. This has led the software literature and software development to be biased towards science and technology. These scientifically oriented people have been trying hard to hammer the software crisis with a collection of mathematical and technical tools, for instance software engineering, software science and software physics. Treating software development as equivalent to the blueprints of construction engineering and electronic circuit diagrams further supports this bias.

The proliferation of scientific and technical persons in the computer industry can be attributed to the involvement of electrical/electronics engineers, in the manufacture and maintenance of the hardware. Moreover, automated computing was regarded as an off-shoot of mathematics during the evolutionary stages of automated computing. In the early days, software was developed mainly for accounting and financial purposes, which was largely mathematically oriented. Therefore, the job of programming was also entrusted to mathematicians and engineers. With the acceptance of computer as a useful tool in every walk of life, a host of large, complex and non-scientific programs were

required. Software was therefore no more a bunch of formulas, but was something encompassing the entire body of knowledge. The technological innovations of these technical persons certainly benefitted the hardware but the cheaper and more powerful hardware demanded equally powerful software. With the knowledge limited to the technical aspects, they were not able to provide the large amount of interdisciplinary software. The declining costs of hardware made it affordable to many organizations. To make the hardware function, increasing amounts of software was required, which was not readily available. Thus, demand exceeded supply causing disequilibrium in the market forces which led to the software crisis.

In every time of crisis, every time of troubles, prophets have come roaring out of the desert, preaching baptism and repentance of sins. The software case is no different. Even today, various groups are taken to the high places and told that salvation is found only in the use of a new high order language, structured programming, software tools, requirements specifications languages, proofs of correctness etc. And the desert is littered with the bones of those who believed and followed. [Ref. 1]. Several books and magazine articles have appeared in the recent years chronicling the recognition of a "software crisis" in the late 1960's and subsequent attempts to deal with it [Ref. 2].

As an attempt to overcome this crisis, several techniques have been proposed. Systems developers are reluctant to use these techniques both because their usefulness has not been proven for programs with stringent resource limitations and because there are no fully worked-out examples of some of these [Ref. 3]. However, a few of these techniques have proved to be useful in developing software as well as in promoting further research.

B. APPROACH

Life is a flowing stream. Some people climb aboard a raft and float comfortably downstream, relaxing and enjoying the scene. Others paddle furiously upstream, determined to explore what is there. [Ref. 4]. This is an upstream thesis. It is expected to be controversial because it takes the difficult approach of solving the software crisis, rather than accepting the mere existence of the crisis.

This thesis is about the crisis being faced by the software--the programs that are needed to make the computer perform its intended tasks. It is intended to be a guide for the managers, and is therefore in purely non-technical language. Buzzwords and jargon have been avoided to a large extent, except when their use was extremely necessary for the sake of clarity. It is based on hammering the software crisis with a common sense approach of management, rather than with complicated equations and formulas. Technical tools and techniques are therefore not discussed in this thesis. Software has not been considered as something special, mystical, or unique. Instead, it has been treated like any other product. Being a product, all the managerial theories and practices are applicable to it. It needs planning, control, and project management techniques such as PERT and CPM. However, they are not discussed in this thesis. The reason for this is that it is not the scarcity of techniques or tools for software development, or tools for project management, or planning and control at the organization level, which are responsible for the crisis. On the other hand, it is the lack of managerial attention at the macro level, which is plaguing the software industry. Therefore, the focus of this thesis is on the management of issues for which the software industry as a whole is responsible. There are plenty of software development tools,

techniques, and methodologies available for use; and there are many planning and control, and project management tools that can be utilized. In fact, these are being used by almost every organization dealing with software, but the crisis still remains. It is therefore the macro issues which need to be looked into, and it is the problems at the software industry level which need to be rectified. Unless these are remedied, no number of tools and techniques, how sophisticated they may be, can take the software out of the crisis.

The terms programmer, software engineer, software developer, software person, software practitioner and software professional are used at different places in the text. They are used just to highlight that all these terms can be encountered by a manager in the real world. They all are, however, synonymous.

II. SOFTWARE CRISIS

A. WHAT IS THE SOFTWARE CRISIS?

The software crisis refers to a set of problems that are encountered in the development of computer software. The problems are not limited to software that does not function properly. Rather the software crisis includes problems attached with the development of software, maintenance of the mammoth amount of software, and keeping pace with the ever-increasing demand of software. The software crisis is characterized by many problems: Schedules and cost estimates are often grossly inaccurate, cost overruns of an order of magnitude have been experienced, schedules slip by months or years and software quality is often suspect. [Ref. 5].

Every engineering discipline has its collapses but they occur more frequently in the area of software development. The collapse of a building or a bridge during construction is a newsworthy event because it occurs rarely. Another point is that whenever these errors are discovered, the perpetrator is expected, and is normally legally obliged to make amends. Major collapses are so frequent in the software field that almost none of them receive much attention. They are accepted as the norm and so the perpetrator is not expected to make amends. Moreover, he cannot usually be identified. [Ref. 7].

According to an established rule of thumb in software production, testing accounts for about half of the development effort of a typical software system [Ref. 6]. The remaining half is divided equally between program design and coding. After development is pronounced complete, considerable additional cost--often more than the development cost,

is incurred during the system's lifetime for maintenance (i.e. finding and correcting mistakes not found during testing, implementing design changes and finding and correcting mistakes introduced thereby). The large amount of effort expended in testing should be a clear signal that something is fundamentally wrong with our approach to software development. However, this message is not apparently getting through to the concerned people. [Ref. 7].

B. CAUSES OF SOFTWARE CRISIS

Why do we have this software crisis? There are several reasons for this.

1. Disequilibrium

One reason which has already been mentioned is that due to technological breakthroughs in the mature hardware industry, the hardware costs are going down. As the hardware is becoming more affordable, software is becoming more complex and expensive. Cheaper, affordable, and better hardware demands more and more of software, which is scarce. Demand has far exceeded the supply causing disequilibrium in the market.

2. Shortage of Software Practitioners

Another reason is that there are too few programmers and they are not as good as they should be, as will be evident from the discussion on variances in programmers' productivity, in Chapter IV. Right now, it appears that the software industry needs every living, breathing programmer it can get, almost regardless of quality. While marginally qualified practitioners are normally squeezed out of a field by economic and competitive forces, these forces are more than counteracted in today's software market by the large

gap between supply and demand. Marginal practitioners are seldom forced out of the software market. They are instead moved to another employer, usually with a raise in salary. Thus, we have simultaneously a shortage of quantity and quality. This shortage stems, basically, from a bottleneck in the educational process. It is customary that a prospective hardware designer have a sound education. If he does not, he is expected to demonstrate that he possesses equivalent knowledge and experience. Lacking this, he may be engaged as a technician or designer's assistant, but not in a designing capacity. In the software field the situation is different. Any previous experience, almost regardless of quality and length, is implicitly assumed to be a more than adequate preparation for designing software systems. Seldom is a prospective software system designer expected to have fulfilled any particular formal educational requirements. While the persons selecting these designers are not really satisfied with the results, they do not, in their opinion have any other choice. Truly qualified software designers are simply not available in the quantities needed. The bottleneck, caused fundamentally by the rapid growth of the computer field, is aggravated by concentration on short term benefits. To obtain maximum benefit now, valuable resources are being diverted from the education of future programmers. Potential teachers go to the industry or are doing research; potential students are deployed as poorly prepared programmers instead of as good software engineers. [Ref. 7].

The shortage of qualified software practitioners is further aggravated by frequent conversions of a technical nature. Considerable manpower is required to convert from one computer system to another, from one operating system to another, from batch to on-line operation, from traditional file management systems to data base management systems, etc. While most such conversions are motivated by the

expectation of increased productivity, too frequently they consume more resources than they free. [Ref. 7].

3. Managerial Error

Many failures of software projects are attributed to managerial error. Management failures result from setting of unrealistic goals and expectations, over-estimation of ability of the organization to design, develop, implement and absorb software systems, and employing underqualified people on software projects. Good advice based on sound knowledge of the technical possibilities and limitations is not always available. When it is, the manager cannot always recognize good and bad advice as such, and to distinguish between the two. When faced with a choice between foregoing a software system because qualified developers are not available or trying to develop it with underqualified persons, the decision process is often dominated by the hope that this time everything will work out well, somehow. Usually, it does not and as a result the scarce resources are wasted.

4. Trial and Error

In every engineering discipline, trial and error also has its place, but only when the designer is knowingly and intentionally working in new areas, "pushing the state-of-the-art" as it is sometimes called. In such cases, the trial and error approach normally takes the form of scientific experimentation. Experiments are designed to yield answers to the open questions, to discriminate between alternative hypothesis, to extend the limiting frontiers of knowledge. The risks are consciously accepted and appropriate precautions are taken. Trial and error is not acceptable when the designer is working in areas in which his own personal expertise is lacking and is not up to the

state-of-the-art. In such a case, the engineer is expected to familiarize himself with the relevant literature and accumulated experience of others before embarking into what is, for him, a new territory. To do otherwise is considered irresponsible. This attitude is not the norm in software development. Experiments designed to yield specific information needed by the designer are not common. Often, the software developer makes use of the "trial and error" approach in its crudest form instead of consulting the existing literature and other competent people. In fact, the programmer is usually unaware of the limitations in his knowledge and experience until an unexpected failure occurs in his program.

5. Unending Search for Technical Tools

Another important factor leading to the software crisis is that instead of emphasizing the nature of software development, emphasis has been on unorganized and confusing technical details. A tremendous amount of effort has been spent in finding the right kit bag of tools which could solve all the software problems. None of these tools such as structured programming, modular design, and top-down coding, have helped in overcoming the crises. They have instead, succeeded in confusing the programmers. The situation is that a promising new tool is proposed, but when applied it fails to solve the problem. After much discussion, the proponents announce that the tool is fine, but is not being applied properly. The search then begins for the right technique for applying the tool and the vicious cycle begins again. No one tries to understand that magical tools for software development do not and will not exist. [Ref. 7].

C. IMPACT OF SOFTWARE CRISIS

Software is becoming increasingly complex and costly. The annual cost of software in the United States in 1980 was around $40 billion, or about 2% of the gross national product, which is expected to grow to 13% by 1990 [Ref. 8]. The growth rate of software is greater than the economy in general. As compared to hardware, the software costs are continuing to rise as shown in Figure 2.1 [Ref. 9].

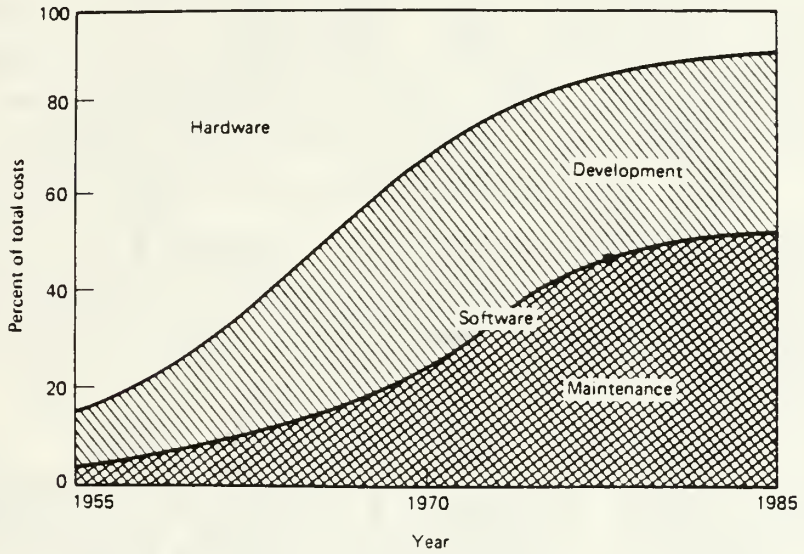

Figure 2.1 Hardware/Software Cost Trends

Due to the trend exhibited in this figure, the impact of software, while making capital investment decisions for computer systems, is much greater than hardware.

An important impact of software crisis is directly on the finances. Software-hardware costs ratio has become lop-sided as is evident from Figure 2.1. It is not uncommon to find that 90% of the total system costs in many

organizations are software [Ref. 10]. These are only the direct costs of software. Indirect costs are even bigger. Delay in the delivery of software delays the delivery of the entire computer system. This in turn, entails additional costs for the organization. Return on investment of huge amounts in the system is delayed and pay-back period is extended. Until the system is delivered, the tasks which are intended to be automated continue to be performed manually, which affects the productivity and profitability of the organization. If opportunity costs of not reaping the benefits of automation for a long period are considered, the indirect costs go much higher. Furthermore, if substandard software is accepted, all the possible and desired benefits are not obtained even after incurring tremendous costs. Finally, there are social costs of the organization's inability to provide the required benefits to the society because of software. Boehm illustrates this phenomenon with the example of software development for a large defense system [Ref. 10].

> It (computer system) was planned to have an operational lifetime of seven years and a total cost of about $1.4 billion--or about $200 million a year worth of capability. However, a six-month software delay caused a six-month delay in making the system available to the user, who thus lost about $100 million worth of needed capability--about 50 times the direct cost of $2 million for the additional software effort. Moreover, in order to keep the software from causing further delays, several important functions were not provided in the initial delivery to the user.

Computers and its software are having a deeper and deeper impact on personal lives. More and more of the activities, such as personal records, bank accounts, traffic control, and medical services are being entrusted to computers and its software. Tassel, in his book "The Compleat Computer", gives abstract from an article published earlier in "Newsweek." [Ref. 11] This article presents a

good picture of the impact of computers in the future. It states:

> Welcome! Always glad to show someone from the early '80s around the place. The biggest change, of course, is the smart machines (computer)--they're all around us. No need to be alarmed, they're very friendly. Can't imagine how you lived without them. The telephone, dear old thing, is giving a steady busy signal to a bill collector I'm avoiding. Unless he starts calling from a new number my phone doesn't know, he'll never get through. TURN OFF! Excuse me for shouting--almost forgot the bedroom television was on. Let's see, anything else before we go? The oven already knows the menu for tonight and the kitchen robot will mix us a mean martini. Guess we're ready. Oh no, you won't need a key. We'll just program the lock to recognize your voice and let you in whenever you want.
> A revolution is under way. Most Americans are already well aware of the gee-whiz gadgetry that is emerging, in rapidly accelerating bursts, from the world's high-technology laboratories. But most of us perceive only dimly how pervasive and profound the changes of the next twenty years will be. We are at the dawn of the era of the smart machine--an "information age" that will change forever the way an entire nation works, plays, travels and even thinks. Just as the industrial revolution dramatically expanded the strength of man's muscles and the reach of his hand, so the smart-machine revolution will magnify the power of his brain. But unlike industrial revolution, which depended on finite resources such as iron and oil, the new information age will be fired by a seemingly limitless resource--the inexhaustible supply of knowledge itself. Even computer scientists, who best understand the galloping technology and it's potential, are wonderstruck by its implications. "It is really awesome," says L.C.Thomas of Bell Laboratories.
> Eventually, for example, they will make possible the full automation of many factories,, displacing millions of blue-collar workers with a new "steel-collar" class. Even office workers will feel the crunch, as smart machines do more and more of the clerical work. Traditional businesses such as television, networks, and publishing companies will encounter new competition as programmers and advertisers beam information directly into the consumer's home.

The picture presented above is a small subset of what the computers are capable of doing and what they eventually will do. If there is any delay in application of computer for the benefit of mankind, it will be because of the software and not the hardware. Software crisis is impeding the progress in applying computer systems to many tasks that are possible and socially desirable. The social costs of the

delay in such applications will soon be high. From a simple economic standpoint, the major impact of software crisis is on the society, which is due to a considerable gap between supply and demand in the software market. Customers are paying unnecessarily high costs for software and its use and they are not getting the maximum benefits. Avoidable costs are also being incurred which are due to errors and failures. Such costs are sometimes shifted unfairly onto persons who are not responsible for the errors and failures that caused them and therefore cannot protect themselves from the consequences of such errors and failures. In many situations the customers, apart from incurring these avoidable costs, loose time when the mistakes are being corrected and are inconvenienced. Even when the costs to any particular person are low, the total economic loss can be high when the many persons who are so affected, are considered.

When incidents with consequences as serious as those mentioned above begin to occur with considerable frequency, a public reaction can be expected. It may consist of calls - some rational, some emotional - for legal and political action. The result will be some combination of restricting and curtailing new developments based on computer technology and social, political, legal and economic pressure to improve the quality of software products and the capabilities of software producers. Such public reactions are not a new thing. They have already occured with other engineering fields. For instance, nuclear reactors for electric power generation and proposed nuclear fuel reprocessing plants have been the targets of demonstrations and legal actions by groups of citizens in several countries.

These pressures to improve the quality of software products and the capabilities of the software producers can also be expected to have a restrictive effect on new applications of computer technology. The introduction of legal liability

for damages resulting from the the effects of software errors would force software suppliers to take the correctness of their products more seriously. This may lead them to employ only those programmers who are capable of producing correspondingly reliable software. The registration and licensing of software engineers will be proposed more frequently and will certainly be considered more seriously in the future. If the supply of appropriately qualified software engineers is not increased substantially, software output will be restricted. [Ref. 7].

The tremendous social and economic impact of software is posing a great challenge for the managers--a challenge to eliminate the crisis and to put software to productive use.

III. NATURE OF SOFTWARE DEVELOPMENT

A major cause of software crisis is that software development has not been understood. Some of the software managers and practitioners are not clear about what software development really is. It is being related to almost every existing field of knowledge in the pursuit of making the theories and principles of that particular field applicable to software development. Software Science and Software Physics are the well-known and mostly unsuccessful examples of such relationships. If this practice continues, soon one will find software chemistry and software mathematics, and there is no end to such relationships. This state of affairs of trial and error is not encouraging. The nature of software development must be decided, so that the future research effort is put in the right direction.

A. SOFTWARE DEVELOPMENT AS A CRAFT

Is software development an engineering discipline or a craft? This question is crucial for effective management of software development. To answer this question we must first understand what is an engineering discipline and how it differs from craftsmanship.

At the first instance, the time before the emergence of engineering methods to the age of master craftsman needs to be looked into. Only then realistic comparison can be made with engineering. The engineering achievements of the master craftsmen were extraordinary. They created many excellent buildings, bridges, ships, furniture and many other remarkable things which are unmatched in more recent times. With the materials and tools available then, no engineer could

have achieved more. These primitive craftsmen were in fact, good engineers. Therefore, an engineer is distinguished from a craftsman because of his tools and methods, and not because of his achievements as an engineer. The craftsman has full knowledge of what he is going to build and knows how to build it. He does not require elaborate plans, scaled blueprints, exact quantities, careful measurements, delivery schedules, progress charts and cost estimates. When he tries to make something, he succeeds because he knows how to make it and his customer knows what to expect. If by chance something goes wrong, he knows how to adapt his work or his design to compensate for the error. Finally, his product works, provides good service and lasts for a considerable time. If it doesn't, then it merely indicates that the craftsman was not such a master as he thought he was and so he would not get the next job. So, "survival of the fittest" was the rule for a long time i.e. only the fittest craftsmen and the fittest designs survived.

A similar kind of situation can be found in the software arena. A software developer starts with a description of what his client thinks he wants. However, the description is so imprecise, inconsistent and even inconstant that it can serve only as a rough diagram and not as a firm plan for implementation. Nonetheless, a good programmer knows how to proceed. He seems to have an intuitive grasp of his programming language and an ingrained feeling for what his operating system can be made to do. He starts writing and testing his code, and when it is all finished, it all miraculously fits together and works. If anything goes wrong, he hacks a bit at his already written code, modifies his plans a bit, and after some delay, delivers his product. If it is not exactly what his client wanted, he can continue to hack until the client is satisfied, or more often, he gets tired of waiting. If the product never works at all, or is too

inefficient or expensive to put into use, nothing happens to the programmer. Unlike the traditional craftsman, he does not get eliminated from the market. The reason for this is that there are less programmers available in the market.

The method of training of the craftsman is also distinctive. No formal instruction in reading, writing or arithmetic is required. A young boy would be apprenticed for several years to a master, and serve as his drudge, assistant and a whipping boy. In return, he would have the privilege of watching the master at work. At the end of a satisfactory apprenticeship, he might be worthy of employment as a paid assistant. After an even longer time, he might become an independant craftsman and hand on the craft to a new generation of apprentices.

This method of training seems to be similar to the methods of training of software developers. Reading, writing and other educational knowledge is not considered important and relevant for the software developer. After a few weeks acquaintance with the esoteric mysteries of some standard programming language, he is thrust into a team engaged in some half-finished project. Some small and unimportant part of the project is allocated to him. When the project is complete, or even before, the experienced members of the team go off to start a new project, and he is left behind on care and maintenance duties. If he is lucky as well as wise, he may learn the software development process by trial and error and by watching the other experienced programmers. [Ref. 12].

It is apparent from the above discussion that treating software development as a craft is a major cause of the problems existing in the software arena. This approach may be all right for simple and trivial projects, but is certainly not appropriate for large and complex projects which require much more than merely the skills and experience of a craftsman.

B. SOFTWARE ENGINEERING

Since the craftsman approach is not suitable for the software of today and tomorrow, it is proposed that the activity of software development is by nature an engineering discipline which is not generally regarded as such in the society, today. Some of the most serious consequences of our current non-engineering approach to programming are: [Ref. 8]

- disappointing and shoddy products, often containing simple errors of a fundamental nature.
- Unnecessarily low productivity.
- diversion of a tremendous amount of effort to unproductive tasks.
- frequent failures of such size that major projects must be aborted at a later stage of development.
- generation of fear, confusion, frustration and misunderstanding among direct and indirect users of computer based systems.

The solution therefore lies in going for an engineering approach; but the question is, is programming an engineering discipline? In 1828, on the granting of a charter to the Institution of Civil Engineers, Thomas Tredgold defined Civil Engineering as "the art of directing the great sources of power in Nature to the use and convenience of man." A computer is not a natural force, but its raw computational power outstrips the mere human calculating ability even more than the steam engine outstrips the puny muscular strength of man. So, this can be taken as the definition of the ideals and objectives of Software Engineering: to direct the great computational power of electronic digital computers to the use and convenience of man.

So far as engineering in general is concerned, one can define engineering as those fields of activities which are

concerned with applying the physical laws of matter and energy to the construction and operation of useful machines, buildings, bridges etc. While this was an adequate definition a century ago, it is too restrictive today. It seems to miss, for example, the essence of that part of electrical engineering concerned with electronics. While matter and energy are necessary aspects of the implementation of electronic devices and systems, the purely mathematical aspects of signal processing would seem to be of more fundamental importance. The abstract aspects of the various building blocks used by the electronics engineer and the manner in which he interconnects them to form a system with characteristics different from those of its constituent parts seem somehow to be more essential than the physical embodiment of those elements and systems.

In deciding whether programming is an engineering discipline, the following questions must be considered:

1. Does a significant body of scientific and mathematical knowledge exist which is relevant to programming?
2. Has the programmer mastered a substantial part of that body of knowledge?
3. Does the programmer actually make use of this knowledge while performing his work?
4. Does the final output (software) take an identifiable, tangible form?
5. Is the software produced of practical value?

To answer the first question, whether the body of knowledge relevant to programming is significant and whether a substantial part of it has been mastered by any particular programmer are, of course, subjective judgements. To make these judgements, it is useful to draw comparisons with other accepted engineering fields. We can ask if the body of scientific and mathematical knowledge relevant to

programming is similar in character and size to that relevant to accepted engineering disciplines. On going through the professional literature, we find that the body of scientific and mathematical knowledge relevant to programming has become qualitatively and quantitatively comparable to other engineering disciplines. Before 1960, this this was probably not true. Around 1970, the point could be argued. Today, it is true.

The answer to the second question is somewhat in negative. Few programmers have acquired formal academic education in software engineering. The others have tried to acquire similar knowledge through other academic degrees, short courses and experience. At the moment, most of the programmers cannot be classified as engineers so far as formal educational standards are concerned. However, as proposed in the next chapter, it is possible to overcome this setback.

The third question should cause a little controversy. Most programmers do regularly use in their work much of the relevant computer science knowledge they have. They may also make use of a larger fraction of their store of professional knowledge in their daily work than engineers in other disciplines typically do. They could also use much more of software development knowledge, provided they have the opportunity to acquire it.

The fourth question can also be answered affirmatively. A finished piece of software takes on several identifiable, tangible forms: printed listings, magnetic recordings, electronically stored patterns, video displays as well as various types of documents intended for human readers. The behavior exhibited by a software system (or more precisely, by a computer executing the software) can be observed, tested and measured.

While not all software, after it has been produced, has any practical value, much of it must be of considerable practical value--otherwise we would not expend ever increasing amounts of effort to produce more and more. Almost all, if not all, software produced or attempted was at least originally intended to have practical value, that is, to satisfy some real need. Even the recent wave of game software for microcomputer systems must be recognized as satisfying a demand for entertainment and therefore as having practical value. Much of it certainly has economic value.

Thus, after answering the above five questions, it is evident that programming is an engineering discipline. However, few writers and professional groups have recognized it as such. The term "software engineering" was used, more in a provocative than in a descriptive manner, as early as 1968, when the NATO Science Committee sponsored a conference in Europe on that subject. The term "software engineering" was chosen as the title of the conference to express the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering. Since 1975, the Institute of Electrical and Electronics Engineers (IEEE) has been publishing a journal entitled "Transactions on Software Engineering". Also, since the mid-1970's, several professional societies have sponsored various conferences and symposia with titles including the term "Software Engineering".The Association for Computing Machinery (ACM) has founded a Special Interest Group on Software Engineering (SIGSOFT).

While a definite trend toward the recognition of programming as an engineering discipline can be discerned in the professional, technical and trade literature, this trend is not a strong one. Such recognition has not yet become

widespread by any means. Programming is probably more widely recognized as an engineering discipline in academic computer science circles and among graduates of computer science programs than anywhere else. Such recognition is probably less pronounced among software houses and other software producers. Most purchasers and users of software products would undoubtedly respond to the suggestion that programming is an engineering discipline with an unbelieving smile. While they might wish that this were the case, and might feel that it should be the case, few, if any, would agree that this is the case today. Only few people connected with the software industry have ever really given the matter any serious thought at all. [Ref. 7]. It is high time that managers in the software industry give full recognition to software as an engineering discipline and direct future research and development in this direction.

IV. SOLUTIONS

A. ALTERNATIVES

On examining the serious impact of software crisis on every walk of life, it seems prudent to find ways to overcome it. The following alternatives will be discussed in an attempt to come up with viable solutions to the software crisis.

1. Status Quo.
2. The Technical Approach.
3. The Missing Link.

1. Status Quo

A computer is such a useful tool that even when sloppily applied by beginners and amateurs, the net benefit--after due consideration of the collapses--is still great. Since the benefits are so great, one might ask, "What is really so bad about the current state of affairs? Large quantities of software are being produced which is of considerable value to its users. While this situation prevails, there is really no problem." While the software industry can be proud of the abilities acquired and of positive results achieved, it must be aware of unjustified self-laudation and its likely consequences. The success of software should not be interpreted to mean that a good job is being done, but that more or less by accident, the industry has stumbled onto a good thing. The damage done by software failures in the past was almost always reversible; in the future, it is likely that more and more irreversible damage will be done if serious effort is not applied to change the status-quo. [Ref. 7].

An argument in favor of maintaining the status quo is that attempting to seek perfection is not really necessary in software. If a bug in software affects the system once in several years and causes, say 1% error in a non-critical result, the intuitive conclusion one draws is that no attempt should be made to achieve perfection. However, the question arises that how will it be possible to ascertain that an imperfect program contains only minor errors. Therefore, logically, it should be endeavored to minimize errors to the maximum extent, to foster confidence in the program. Though perfection is not practically possible, trying to achieve it, is a path towards gaining confidence in programs.

Another argument is that only that software should be developed, which is humanly possible. This scaling down of aspirations can eliminate software problems. This argument also does not carry weight because it is a human instinct to look forward and grasp things which are beyond reach i.e. to never be satisfied with the status quo. [Ref. 13]. Moreover, what is there to be satisfied with? The problems are immense and the impact of software is so great on all human activities that maintaining the status-quo does not appear to be a plausible alternative at all.

2. The Technical Approach

Several concepts, technical tools and methodologies are being proposed without analyzing them in pragmatic depth. Structured programming, Top-down coding, modular design, HIPO (Hierarchy plus input-process-output), are just a few examples of such concepts. No one ever does cost-benefit analysis of these concepts, before proposing them for implementation. It has never been ascertained if elimination of "GO TOs" is worth the price of pre-compilers, or if "top-down coding" is really practicable. [Ref. 14].

The person suggesting a new concept is confident in his mind and makes others believe that this one will somehow work, and the world will be relieved of the software crisis. When it doesn't work, the argument is given that it was not applied properly. Even after proper application, not much fruitful results are achieved. Then the software world believes that this was not the right concept, and waits for the right one to to be discovered to solve their problems. The next concept is however, no better than the previous ones.

Having tried so many concepts, it is time that the software world understands that emergence of omnipotent, magical, and panacean concepts, tools, and methodologies, to wipe away the software problems, is not likely.

3. The Missing Link

Having seen that the above three alternatives are not feasible, active steps must be taken to change the status-quo. The most important change which must be made is to "get down to basics." On getting down to basics, it becomes evident that a vital link is missing. Several software development techniques and tools are available and much research is going on in this direction. What is missing is the link of management which is so important a link that it forms the base of all software activity taking place in any part of the world. Yet management has been overlooked, due to the dominance of scientists and technicians in the software industry.

The management discipline can be divided into project management, general management and higher-level management. Project management is involved with the day-to-day activities of a project. General management is at a level above a particular project organization. Higher-level management is at a level further above, going beyond the

boundaries of organizations and falling in the domain of the industry as a whole. Even though the first two categories have not been totally exploited and made fully applicable, it is heartening to observe that at least there is a considerable awareness of these among the software developers and managers. It is the absence of higher-level management, which is plaguing the software industry. To overcome the crisis, it is imperative to strengthen the base of the software industry. Hence, the ensuing discussion will be devoted to this high leverage category of management.

B. EDUCATION

Just as a fine surgical instrument is of value only when in the hands of a professionally trained and skilled surgeon, so are the tools and techniques of software technology of full value only when applied by professionally trained and skilled software engineers. Much more emphasis should be placed on building the educational base of the software developers. Much less emphasis should be placed on the search for magical, panacean tools and techniques.

If one traces the productive ancestry of any software system, ultimately the human brain will be found to be its original progenitor. The human brain is, therefore, the primary factor of production, of software. In hardware, there is much emphasis on the factors of production like technology and technological skills, but unfortunately much emphasis is not found on the software's factors of production. A software system can be only as good as its developers' intellects are capable of making it. To improve the quality of software, the quality of the intellects which produce it must, therefore, be improved. To increase the quantitative capacity to create software, the requisite knowledge and skills must be transferred to a great number

of human minds. In other words, to improve the quality of software, better education must be provided for the software practitioners; to increase the much talked of "programmer productivity", more practitioners must be educated. Providing more and better tools and techniques to practitioners inadequately equipped intellectually to employ them creatively will not solve the software problems.

Only a small fraction of the programmers of today have completed formal academic programs. While some others have acquired comparable knowledge in other ways, many practitioners have not mastered a substantial part of the relevant body of knowledge. Thus, even if one does conclude that programming is an engineering discipline, not all programmers of today can be considered as engineers. When considering the educational needs resulting from the application of software technology, it will be worthwhile to consider the educational paths followed by those persons technically responsible for the application of older technologies to society's various needs. The designers and developers of any product e.g. buildings, ships, aircraft, are all required to have completed a several years of academically oriented, university level course of instruction, before practicing their professions. The situation is different in the software field. Instead of a university education, reliance is on short courses in programming in the hope that the programmers will do wonders on completion of such courses. Only two institutions, Seattle University and Wang Institute, have awarded master's degrees in software engineering. The demand for such University graduates is much more than the supply. Therefore, it is imperative to widely introduce undergraduate and graduate degree courses in software engineering. Software practitioners should also specialize in a particular application area, such as business software, and scientific software, so that they have

adequate knowledge of the system they are developing. Managers must ensure that they are deployed on software development in their area of specialty. Unless concerted efforts are applied in this direction, the gap between supply and demand of software engineers will continue to widen.

It is striking that the developers of computer hardware are academically trained engineers in contrast to the software developers, most of whom do not possess academic degrees in their area of specialty. It in no way implies that all the credit for this state of affairs goes to the hardware people; nor does it mean that the software managers are dummies. It is just that the academic programs of the hardware designers evolved naturally from already well established courses of study in electrical/ electronics engineering. The evolution took place for the most part within the existing large departments of electrical engineering at recognized academic institutions. It was not necessary to found new departments and faculties to establish the organizational base for sound engineering programs in computer hardware development. In the software field, the situation was much different. Academia experienced much greater problems as no natural base existed on which software engineering could grow in a natural way. However, the sad part is that even after about thirty years of software life, the situation has not changed much. [Ref. 7]. A few curricula for graduate and under-graduate degrees in Software Engineering Engineering have been proposed but not widely implemented. A curriculum for Masters degree in Software Engineering was proposed by Peter Freeman in 1978 [Ref. 15]. Seven years have passed but this curriculum is still in the proposal stage. When long delays in the software development development process, are observed, it becomes clear that the people in the software industry have

become accustomed to procrastination, which explains why the curriculum is yet a proposal. In fact, there is need for a much comprehensive curriculum which includes more of managerial courses and application area courses. The proposed curricula are dominated by courses dealing with various tools and techniques of software development, some of which can easily be substituted by managerial courses. The introduction of an equal amount of management courses is also supported by the definition of software engineering, which states that software engineering is the application of sound, established engineering and management principles to the analysis, design, construction, and maintenance of software and its associated documentation.

One of the causes of the software crisis is that instead of emphasizing education for the software engineers, an alternate route has been taken. In an attempt to circumvent the shortage of qualified software engineers, conscious efforts have been made to deskill programming. By creating the impression that programming requires only minimal skills, highly capable persons have been discouraged from becoming programmers and too many with marginal aptitude and marginal educational qualifications have been encouraged to join the rank of programmers. This effect is more pronounced in the European and Asian countries, where until recently it was socially unacceptable to place a college graduate in the position of a programmer. This effect still persists in the under developed countries. It is therefore, understandable that the quality of the software being produced would be low. If aircraft piloting was deskilled, an increase in the number of crashes would certainly be expected; or if civil engineering was deskilled, a drastic increase in the number of collapses would be observed. [Ref. 7]. Deskillment of programming is therefore, not acceptable. Managers must take cognizance of the consequences of this approach and make serious efforts to curtail progress in this direction.

There is no kit bag of tools and techniques which if learned can ensure success of the software engineer. Software engineers must have a university education comprising of courses in business, management, mathematics, and computer hardware and software plus an application area in which they desire to specialize. Education is also vital for the users and the management. They need not be programmers but at least they should have enough knowledge of the important tool they are using as an aid to their day to day activities.If they expect the computer to provide aid to them, they should also learn to provide aid in the development of its software.

C. SHORT-TERM ALTERNATIVE TO EDUCATION

It is understandable that the revision and implementation of software engineering curricula will take considerable time. Meanwhile organizations have only two choices. One is to hire people with degrees in other technical disciplines such as computer science, electrical engineering or mathematics. The other choice is to lure software engineers from other organizations. None of these choices are satisfactory. Persons with degrees in other disciplines will be lacking in the requisite knowledge required of a software engineer. Robbing Peter to pay Paul is also not a wise solution, as it only shifts the problem to other organization. The problem of the industry still remains.

McGill gives a third choice of cross-training of own experienced engineers in the discipline of software engineering, by various organizations. [Ref. 16]. This training can be conducted in-house with the help of qualified and experienced software engineers and managers. Such training should cover managerial as well as technical subjects deemed essential for effective and efficient

software development. This is however, not a substitute for education, but merely a short-term alternative.

D. SOFTWARE RECOGNITION

Managers regard software as a coat of paint on the hardware. What they fail to see is that the hardware to which the paint is applied is like a street sign. It is useless without the coat of paint. Furthermore, the paint on the street sign must be durable, highly visible, in sharply contrasting colors, and perhaps reflective. The cost of the paint applied to the sign is low, the time necessary to apply it is short, and it can be changed easily. Moreover, the research and development that made the paint durable and visible and made the graphics of the sign meaningful, is also substantial. The same paint and graphics applied to a piece of cardboard convey most of the sign's utility without any metal. The intrinsic value of the sign, therefore, is not in the hardware at all, but in the software. [Ref. 17].

From the earliest days of computing, the purchaser has subconsciously felt that what he was really buying was the hardware. The software was a minor accessory. This attitude reflected the cost structure of computer systems upto the early 1950's, reasonably well. Beginning in the late 1950's and during the 1960's and 1970's, however, the industry's pricing policy did not reflect this cost structure and the typical computer user still had the impression that he was buying the valuable hardware and getting the cheap software as an accessory.

The users must realize that the solution to their problems lies not in the hardware but in the software. When a need for computer arises, they must determine the requirement of software. Only then should they select the hardware required to execute the software. They should really buy the

software and treat hardware as the accessory. This is easier said than done. Even if the user consciously recognizes this fact, his subconscious, however, will still perceive the hardware to be the object being purchased and the software to be an unimportant accessory. There is an understandable reason for this psychological effect. To every human being, even software experts, computer hardware is more tangible than software. The choice of the words "hardware" and "software" can be attributed to this common human perception. The more tangible thing, having more substance, is perceived subjectively to be of greater value. While software takes a tangible form, its essence is perceived as largely intangible and hence of less intrinsic value. No matter how difficult it may be, this deeply ingrained aspect of human psychology must be overcome to subconsciously and genuinly recognize that software is the actual good being purchased.

Software needs to be recognized not only by the entire body of users but also by the industry at large. The importance of software recognition is essential to demand optimum educational standards of software practitioners and to induce the academia to take cognizance of this vital demand.

E. PROFESSIONALIZATION

Should software practitioners be professionalized? This is an important question to be answered. Before coming to any conclusions, we must have an understanding of the difference between a professional and a non-professional. A main difference relates to what they guarantee. The professional never guarantees results, he guarantees instead a certain level of personal qualification for performing the service offered. For example, physicians, lawyers, architects and engineers do not guarantee success. For the non-professional, the situation is reversed. He does not

make any formal representation regarding his personal qualifications or abilities; instead he guarantees that the services rendered or goods delivered will satisfy previously agreed standards or specifications.

At present, software is a no man's land. Neither a guarantee of the correctness of the software is given nor any formal representations regarding the qualifications of the software developers are made. This is obviously unfair to the purchaser who has a right to demand some guarantee of the quality of the software he is obtaining. The typical software developer neither has the qualifications of a professional level to guarantee the product nor is he capable of developing a complex software system in which he can have enough confidence that he can guarantee it's performance. Thus, the inability to guarantee the software is due in both instances to the inadequate qualifications of the software developer.

As a step towards overcoming the software crisis, the qualifications of software practitioners must be improved so that they are able to guarantee something. We have to make a choice between professionalization and non-professionalization. Since most of the complex software being developed is not well within the state-of-the-art, professionalization seems to be a plausible alternative. Moreover if the electrical, mechanical, civil and other engineers are professionals, why can't software engineers be professionals. Apart from providing a solution to the guarantee dilemma, professionalization will help enhance the abilities and skills of the software engineers. For this purpose a professional body must be formed on the same lines as other professional bodies like Institute of Electrical and Electronics Engineering (IEEE), Data Processing Management Association (DPMA), American Medical Association (AMA), Institute of Management Accounting (IMA), etc.

Software Engineering should be taken out of the domain of IEEE and the Institute of Software Engineers (ISE) be established. ISE can lay down standards and code of ethics for the software engineers. It can also carry out certification of the software engineers based on qualifying on an examination encompassing all the required disciplines. Besides the long-term benefits, ISE can also provide a short-term solution to the problem of inadequate qualifications of the software engineers. Finally, it can foster wide-spread recognition of the importance of software and thus facilitate elimination of the software crisis.

F. PROGRAMMER'S PRODUCTIVITY

1. Why Increase Programmer's Productivity?

Mankind should not strive to do everything which is technologically possible. This statement stems from the following two propositions.

1. Man should strive to achieve the technological capability to do whatever he decides he should do.
2. Man should not do everything which his technological capability enables him to do.

The application of the first proposition places a responsibility on managers to strive to improve the abilities of programmers to create software which does what has been specified and does not do anything else which could cause injury, loss or inconvenience to the society. This proposition also places a responsibility on managers to strive to improve the programmers productivity. While productivity does not effect what one can in principle do with a given technology, it does effect, what society can in practice do with it. [Ref. 7]. Hence the importance and need for increasing the programmers productivity.

2. Factors Affecting Programmer's Productivity

While all the factors discussed earlier affect the programmers productivity, there are certain other factors which must be taken into consideration. One factor crucial to productivity is that there are tremendous differences between programmers. In 1968, three SDC researchers - Sackman, Erikson and Grant concluded that the most important practical finding of their research was that striking individual differences existed in programmer performance [Ref. 30]. They found that

- Capability to debug differed by factors up to 28-1.
- Capability to code differed by factors up to 25-1.
- Timing efficiency of the resulting program differed by up to 11-1.
- Sizing efficiency differed by up to 6-1.

Schwartz, while studying the problem of developing large software systems came up with similar findings. He found human factors at the heart of the problems and stated "within a group of programmers, there may be an order of magnitude difference in capability" [Ref. 31]. In 1973, Barry Boehm observed productivity variation of 5:1 between individual programmers. In 1978, Myers found [Ref. 32].

> There is a tremendous amount of variability in the individual results. For instance, two people found only one error, but five people found seven errors. The variability among student programmers is generally well known, but the high variability among these highly experienced subjects was somewhat surprising--The detection of individual type of errors varies widely from individual to individual.

Raymond Rubey, while exploring the impact of higher order languages on avionics software, wrote [Ref. 33].

> A programmer having no prior experience wrote a program that was 100% inefficient, while an experienced programmer wrote a version of the same program that was 20% inefficient. Another study reported a 25%

improvement in efficiency with greater programmer experience. Clearly the programmer's experience is a major factor in achieving high efficiency.

3. How to Increase Programmer's Productivity

a. Skills

The problem is evident from the work of these researchers, but the solution is not clear, i.e. How to get the right people? There has been a period of aptitude tests that did not prove successful. There was not much correlation between test scores and performance. The SDC study also substantiated this fact apart from proving that there was not a consistent correlation between class grades and performance, as well. It means that neither the aptitude tests nor the academic performance are true predictors of on-the-job-performance. The challenge, therefore is to improve the individual skills to increase the programmer's productivity. To face the challenge objectively, first, educational standards of the programmers must be improved. If education has no correlation with on-the-job performance, something is wrong with the educational system. If the programmer's efficiency increases with experience, it is an indication that there is plenty of room for improvement in his skills when he graduates. This gap between the quality of education and the skills required by the software industry must be bridged. There is no short-cut to experience, but proper academic programs leading to a degree in software engineering can certainly enhance the skills of the software engineer. The existing programs in computer science or information systems are not bad. These programs are sure to provide good computer scientists or managers to the industry but not good software engineers. One or two courses in programming languages are not sufficient. Six or seven

basic programs cannot make a student a programmer. Much more exposure to programming and to other software development tools and techniques is required.

b. Certification

The second solution to increasing programmer's productivity is certification of software engineers. There is certified data processor exam and the ACM self-assessment program, but their results also show no correlation with on-the-job performance, because the persons who obtain such certification are obviously better data processing or computer professionals but not better software engineers. Therefore, as proposed earlier ISE must be established soon, which should provide certification of software professionals.

c. Right Man for the Right Job

With appropriate education and certification, individual differences in programmers efficiency will be reduced but not totally eliminated. After all programmers are also human beings and they are not meant to be alike. The programmer armed with educational and professional qualifications, needs to be evaluated by the management. On the basis of this on-the-job evaluation and experience, programmers must specialize in the areas of interest in which they have natural aptitude. Some programmers will be better in coding, some will have a natural talent for debugging and some will be good in planning or maintenance. Onus lies on the management to ensure that right man gets the right job, if overall productivity has to be increased.

d. Environment

The programmer of today has a rich variety of projects at hand. The environment in which the programmer works, varies with the projects on which he is involved.

From air-conditioned, carpeted offices to temporary buildings at a far-flung site, there is not and cannot be a "one-only" working environment for the typical programmer.

The idea of providing a good working environment as a step towards increasing programmer's productivity, and thereby overcoming software crisis, sounds trivial. Nevertheless, this apparently trivial aspect has serious implications on the long run. Working environment does affect performance. The famous Hawthorn studies dealing with the effect of lighting on productivity, supports this contention. Now the question arises that what is really a good working environment for a programmer? Unfortunately, the computer literature has not much to say on this subject.

Apart from the normal comfort requirements like lighting, air-conditioning, and heating, there are some aspects of programming which call for unique needs in a working environment. The programmer must understand the "big picture" of the problem he is involved in, and be able to discuss it meaningfully with his management and customers. He must also work at the nitty-gritty level with coding sheets and memory dumps in a foreign language. The working environment should cater for these diverse requirements. In the big picture realm, there should be conference rooms to which the programmer and his superiors may retreat to haggle with a customer, without disturbing others. Conference rooms can also be used for meetings of team members involved in a project. In the nitty-gritty realm, the programmer must have an independant office where he can go for thinking and coding. [Ref. 4] Apart from providing an isolated environment suited for thinking and coding, separate office will also enhance the status of the programmer. After all, he is a human being first, and then a programmer. His needs are therefore, more or less same as any other employee in the organization.

G. BRIDGING THE GAP BETWEEN ACADEMIA AND INDUSTRY

Let us look at the academic and industrial world of software. Researchers in the academic world go for those things which they consider interesting. They are rewarded for this by the recognition of their peers. In the industrial world, researchers go for those things which they consider useful. They are rewarded for this by the recognition of their management.

This conflict of interests is not a minor thing to be ignored. The academician looks with some disdain on things which are merely useful. The industrial person looks with some disdain on things which are merely interesting. Each side is thus making judgements about the basic goals of the other. Actually, the pursuit of interesting work is an ethic to the academician, and the pursuit of useful work is an ethic to the industrial person. Proofs of correctness, requirements language and symbolic execution are examples of things which are interesting but not really useful, and so they interest the academician. Testing, requirements reviews and peer code reviews are useful but not really interesting, and so they interest the industrial person. Where does this dichotomy leads to? Well, it leads to lack of knowledge sharing. The researcher who pursues primarily interesting problems communicates poorly with the researcher who pursues the useful ones.

It is time for academic and industrial people to review their ethics. There is no problem in working on interesting or useful problems as both are legitimate. The problem is disdain on each others part and lack of knowledge sharing.

There are other differences too between the academicians and the industrial people. One major problem is that they don't talk to each other, and even if they do, they will not understand each other. Journal of ACM kind of article will

never be published in Datamation, and even if it does, no one will understand it, as the jargon of academicians are different form the industrial world. Then, one can find mutually exclusive audience in academic computing conferences and industrial computing conferences. Academicians do not understand the immense complexity of the industrial world and industrial people do not understand what the academicians are doing for them.

One more problem is that the practitioners don't write. The "Publish or Perish" syndrome forces academics to write, and write too much. The "Proprietary" syndrome forces the practitioners in the opposite direction. There are brilliant practitioners who can benefit the software industry by sharing their experiences with others. Managers must therefore, encourage them to write, and bring their experiences to the public's knowledge, by remaining within the confidentiality limits of their organization. [Ref. 14].

The gap between both these professionals must be bridged if the computer industry in general and software industry in particular, is desirous of attaining any synergy. One way to bridge this gap is to have greater contact between them through university-industry exchange programs. The practitioners should take regular courses at universities and the academicians should spend regular periods in an industrial environment. This can make the academicians/researchers aware of the needs of the practitioners, and make practitioners capable of meeting the demands of changing technology. [Ref. 18]. Both these factions can gain a lot from each other, which will in turn be beneficial to the industry as a whole. If the academicians and the industrial people join hands and work in the same direstion, there is no doubt that the days of software crisis will be numbered.

H. MANAGEMENT - TECHNOLOGY OVERLAP

In software development, there are both technical and management problems, and there is some overlap between them. Technical problems include coding techniques, design methods and programming standards. Management problems include, for example the aspects dealing with people, like training, motivation, turnover, scheduling of jobs, etc. However, overlap occurs on many items. For instance, documentation of software is both a technical and a management problem.

Generally, documentation is regarded as the thing to be checked to ascertain quality of the software package, which is not correct. Quality can be evaluated only from the listing of the program, as it truly represents a software product. It is the only fully accurate and up-to-date representation of the code which computer executes. Managers are charged with the responsibility of evaluating the quality of the software product, whether they accept it or not. Nonetheless, they tend to avoid reading the program listings because they lack the required technical competence. They like reading english like documentation and evaluating the quality on a wrong basis. Managers who do not read program listings cannot tell whether the documentation is correct or not. Therefore, the programmers take advantage of this handicap of the managers. Programmers are too bored with each others code and testers do not have to look beneath the behavior of the program. Ultimately managers are responsible for the quality and they also do not do the right evaluation. It is not because they do not want to, but it is because they do not have the ability to do so. As a result, the poor software suffers.

Therefore, managers must learn some of the technical aspects of software development. Likewise, the technical persons must learn some of the managerial and organizational

issues, to carry out their job efficiently. There is a word of caution, however. The management-technology overlap emphasizes learning of the required discipline by both the parties. It in no way implies taking over the jobs of each other. A common problem observed in developing software is that competent technical people become managers and spend most of their time dealing with administrative problems at which they are less competent. There are several solutions to this problem. First, technical people should not be promoted to managerial positions, on the pretext of job-recognition. Their career paths should be established within their own specialty. This point is covered in depth in a later chapter. The second solution is the inclusion of managers/administrators in software projects. Finally, team form of leadership, with one person providing technical guidance, and the other handling administrative matters, is another alternative. [Ref. 19]. In no event, the technical people, no matter how competent they are, should be made managers, as they will always make either mediocre or poor managers. Loosing a good technician, and getting a bad manager is not a plausible alternative.

I. REINVENTING THE WHEEL

It has been said that programmers always try to reinvent the wheel. They waste a considerable amount of time in discovering facts which are already known. The only way to avoid this syndrome is to share. If the programmers share each others work, they can increase their productivity and reduce frustration.

Programs can be divided into two broad categories.

- Private.
- Public.

1. Private Programs

Private programs are written by students which do not have to be reliable, portable or documented. Such programs are a kind of one-to-one communication between the programmer and the computer. They are considered as personal objects whose sole owner is the programmer. The tasks of specification, design and implementation, all are undertaken by one person. These programs are discarded soon after they are run.

2. Public Programs

Public programs are usually written by more than one person. They are generally large and complex, and so it is not possible for one person to undertake all the tasks of specification, design, and implementation. They have to be reliable, portable and documented. They usually tend to have a long life.

Programmers are not trained to write public programs. Their training process focuses on writing small private programs. Such type of training creates an attitude that programs are personal property and if the authors can run them, it is a job well done. The students learn to write small programs which do not have any practical utility and are merely for getting to know a certain programming language. Partly, due to time constraints in an academic courses, the programmers get the misconception of writing private programs. These poor programming habits and attitudes, which are ingrained in the programmer by the educational system are hard to change later. They keep thinking that programs are personal property and so cannot be shared. Moreover, building on programs of others is questionable in the academic circles whereas it is common sense in the real world. In the academic world copying is considered wrong,

whereas in the real world, cutting and pasting from other programs to make new ones is expeditious and wise. Sharing is therefore, considered bad by the programmers even when they are confronted with large public programs.

The solution to this "re-inventing the wheel" syndrome is to teach programmers to share. They should be taught to develop large public programs, but the conversion from private to public programs should not be sudden and abrupt. At first, they should be given experience in working with small public programs and then they should be involved in larger programs. Managers should make the programmers understand that any program worth writing will be useful to someone, sooner or later. Therefore, they should make the software reliable at the first instance, keeping others in mind. For this purpose, they should plan for the software to be public, and should design and document the programs and label all outputs. [Ref. 20].

J. TRANSITION FROM LABOR-INTENSIVE TO CAPITAL-INTENSIVE INDUSTRY

Software development is predominantly a labor-intensive activity. Research efforts are going on to automate the development process, but it still remains a manual task. If the situation does not improve, the requirement of software engineers which are already scarce, will increase manifold. Martin estimates that there are presently 300,000 programmers in the United States, and predicts that 28 million programmers will be required ten years hence, unless there are drastic changes in the manner in which software is developed [Ref. 21]. On the other hand, hardware which is capital-intensive, is getting cheaper. IBM 370 equivalent computer has got down to a single card with three microprocessor chips, and it is expected that eventually, it will

be available on a single chip. The software development tasks should therefore, be shifted to hardware, as much as possible. Software crisis is due to increasing amount of automation requiring more, better, and complex software. Automation is the cause of software problems, and it is automation which can reduce these problems.

V. SOFTWARE AS A PRODUCT

A. THE PRODUCT CONCEPT

Products do not just emerge. They are planned and designed to meet certain specifications and are built because management recognizes certain needs which they have the potential to fulfil in a profitable manner.

Products perform useful functions. Moreover, many copies of it can be made, manufactured or replicated so that there is usually a large customer base (users/clients) for the product. Then, they are supported with promotion, training and maintenance.

Another attribute of a product is that it has concrete external specifications. Documentation is usually supplied with the product that describes in detail the functions that it performs and also how they are performed.

Usually a product is part of a system of products and must be assembled and installed either by the end-user or by a specialist representing the supplier. To assist this assembly, additional documentation is provided by the supplier.

If a product is new, complex, or different from other products, then training is often provided. Training can be given in two ways. It can be a self-instruction process, by making use of the manuals provided by the manufacturer. Else, supplier arranges formal classroom instruction, in case the product is highly complex. Training covers what is given in the documentation i.e. how to install the product and how to use the product.

The documentation and training mentioned above covers the external features of the product. Apart from this, there

is other documentation which includes features internal to the product i.e., how the product works, how it was built and what parts it uses. Manufactured products can include manufacturing drawings and specifications, bills of materials, and sources of supply for parts and raw materials. For software it includes specifications, listings, programs, data generators and bills of materials. For maintenance of the product, more internal documentation is required, which includes enough of the principles of operation of the product to satisfy the needs of the person carrying out the maintenance. It also includes specific instructions for trouble-shooting and for repairing common malfunctions. User must also have access to maintenance service or he should be prepared to maintain it himself, throughout the useful life of the product.

In case of a product having a long useful life, the documentation also includes adequate internal description of its operation to allow modification of the product, either by the user himself or by his technical agent. A major example of such attribute is the automotive market. Considerable internal documentation is required by auto enthusiasts to facilitate performance enhancements or to exchange engines. Such documentation is used to tailor or customize the product for a unique use. It is also used to enhance or upgrade the product when design changes come about through experience gained in using the product or when new features are introduced by the manufacturer. Aircraft industry is a good example: worn out engines are often replaced with new designs.

Another important attribute of products is quality control. The product must behave reliably and predictably as described in the documentation, and identical copies of the product must behave in exactly the same manner. [Ref. 17].

If we examine the above attributes of products, we come to the conclusion that almost all these attributes are present in software. Yet, many managers do not treat software as a product. A reason often given is that software cannot be seen or felt. It may be worth noting that products in gaseous form cannot be felt either, but they are still considered as products, for example gas filled in a cylinder can neither be seen or felt, but the fact remains that it is a product.

B. REQUIREMENTS - THE SOREST SPOT

1. Import and Impact

Requirements definition is the most important phase of software development as the entire life of the software product, right from inception to obsolescence depends, on it. A considerable research is going on to find out means to automate the software development process. All the phases of software development can be automated except the requirements definition. Human interface will still be required to determine the needs for the product, and it will thus remain for the most of the part, a manual and labor-intensive process.

It is also a well accepted fact that maintenance occupies a major portion of the software costs. Industry surveys have indicated that 70% of the software costs are attributed to its maintenance, and 40 to 95% of the manpower effort in typical industrial applications occur during the software maintenance [Ref. 22]. Moreover, most of the maintenance is due to changes in the requirements, of which an over-whelming portion is of the avoidable changes.

2. The Users Dilemma

The most common reason of software failure is inadequate requirements definition. Maintenance occupies a major portion of the software costs because the requirements are not fully defined before proceeding with the development of the software. The software developers do not know what their product is required to do. As a result, they produce software which is not what it was meant for. Requirements definition is an essential and most important input to the software development process. Requirements are needed by the developer , but are to be obtained from the users, over whom he has no control. Yet he is responsible. It certainly sounds odd. How can a person be responsible for something over which he does not have any control. It is not only against the basic management principles, it is against common sense. Nevertheless, the fact remains that the software developer is at the mercy of the users.

Users are busy and so is everyone. Software developers have to urge and beg the users to spare some time for the requirements. After considerable effort, the requirements are given, which are incomplete and usually incorrect. Users do not take any interest and do not devote much time to this process on the plea that they are already hung up in their day-to-day activities. More often, they are not aware of what they want the computer system to do for them - they are not cognizant of their needs. In other times, they know what they want but are not able to articulate it. Consequently, they develop the following strategies, which are nicely explained by Laura Scharer [Ref. 23].

a. The Kitchen Sink

This strategy is employed by those users who throw everything into their requirements definition. The

outstanding characteristics of this strategy are exaggeration and a protective overstatement of needs. An overabundance of reports, exception processing, and politically motivated system features are also symptomatic. The Kitchen Sink also provides a marvelous cover-up for the user who doesn't know what he wants but who can bury that fact in the sheer volume of his requests.

b. Smoking

Known also by its full name, "Smoke Gets in Your Eyes," this strategy is practiced by the user who sets up a smokescreen by requesting ten software features, knowing that he really wants only one of them. The nine extra gives him a bargaining power, at least he thinks so. The smoker is usually an experienced user who is consciously manipulating the definition process as opposed to the Kitchen Sink user, who is usually naive in believing that he really needs everything he asks for.

c. The Same Thing

Sometimes a euphemism for the embarrassing words, " I don't know," sometimes a sign of laziness "Just give me the same thing I am getting now," are the hallmarks of this strategy. The latter statement is sometimes qualified in many ways such as "... but more accurately," or "... but more timely," or "... but computerize it. "The user who employs the "Same Thing" strategy is often satisfied that he has told the analysts everything they need to know to proceed. In fact, the only thing the analysts really come to know is that the user is not aware of what his current system does, and that he does not want to take the time for an introspective review of his own functions and problems.

In all of these three strategies, we observe that the requirements are not adequately defined. The more

interesting aspect is that no sooner the requirements are given, they are changed. These changes keep coming even upto the eleventh hour, and the developers keep accepting them whole-heartedly on the plea that "programs must evolve." "I didn't mean that," "Actually I want this," "It was then, I wanted that, but now I want this." These kind of statements are often heard by the software developers during and after the development process. One can hear such comments from the users, minutes before the acceptance phase. Even after acceptance and implementation, these phrases are not too uncommon.

3. The Effects

The above state of affairs leads to schedule slippages, excessive maintenance, costs overruns, and software failures, which is exactly what is known as software crisis. With constant changes the job has to be redone or code modified, which consumes time, causes delays and involves costs. This, therefore makes it difficult for the software developers to adhere to the schedules and remain within budget. Inadequate and incorrect requirements definitions, does not meet the users' requirements, so the software fails to perform the functions desired by the users, or more precisely meant by the users.

Another impact is on software productivity. It appears from the foregoing discussion that software productivity is dependant on the quality of the users. If users are cooperative and knowledgeable, the productivity will be higher and vice versa. Generally, the users do not possess such attributes, and consequently the software productivity suffers.

4. The Answer

It is obvious that we have a dilemma which warrants serious consideration. As mentioned earlier, software is a product. Therefore, this problem can be solved by treating software as similar to any other industrial product. For illustration purposes, let us take the example of automobiles. While manufacturing automobiles, the manufacturer is not required to to go to each user to ascertain what they want. He simply carries out market research, either in-house or through an outside agency. This research establishes the general preference of the users/customers. The automobile is then manufactured keeping in view these preferences and other factors such as economics and technology, into consideration. There is no doubt that that the automobile will not meet the needs of every potential user. However, it is brought to the market and sold. Customers buy it, if it meets most of their needs as compared to any other make available in the market. Thereafter, they either match their needs/preferences with this particular automobile, or they modify it if they feel like. For instance, they may like to modify the engine to make it work with diesel instead of gasoline. There is, however, another alternative, of buying a customized one if the customer's budget permits or if it is considered absolutely necessary. This analogy is not unique for automobiles only, it holds good for any other product.

Software can be treated in the same manner as automobiles, or for that matter any other product. What is meant here is that software should be standardized, as far as possible. It should be developed after obtaining the requirements/preferences of the general body of users. Specific users should then match their needs with the software product, and not the other way around, as is being done

at present. However, it may not always be possible to match own needs with the product. In that case, users can modify the software either in-house or through the manufacturer. This should however, only be done if the need for modification is too pressing. For instance, when a company wants to maintain confidentiality about data and type of business practices being implemented or when a scientific program is required for special purposes, or when there is a change in legal requirements. The trade-off between modifying and matching to the needs versus adapting own needs to the software product, must be examined and evaluated. In other words, software should be modified only if the benefits outweigh the other approach of its adaptation to own needs. It must also be kept in mind that programs have limitations, and they should therefore, be used only for what their capabilities permit. An automobile cannot be driven beyond the maximum speed limit imposed by the manufacturer. Similarly, programs should not be run for purposes not specified by the manufacturer. Another alternative to extensive modifications is to have modules available separately, just like spare parts for any other product. These modules should have the properties of coupling and cohesiveness, so that several modules can be assembled together and a program made.

Modifications should in no way be done for trivial reasons. Modifying an inventory or payroll package, just because a company has different formats of forms, is not advisable. In such a case, the formats of the forms can be changed, if all the information can be provided by the software package. Often modifications will not be required. Moreover, modifications will involve lesser costs as compared to in-house development of a new product or development of a customized product outside. The standardized software product will be much cheaper as compared to the customized one. Therefore, several similar products will be

available in the market for the user to choose from. The user can pick the one which caters for the maximum number of his needs.

There is another advantage of this product approach. Costs of manufacturer of the software product will be spread out and so the customer will pay little price. When software will be cheaper, it will be more affordable. With hardware already being cheaper and affordable, more and more applications of computers will be seen, which will benefit the society as a whole.

If due to peculiar needs, a company decides to have a customized software developed, whether in-house or outside, then the onus of providing accurate, adequate and specific requirements must lie with the management. For this purpose, management must develop sufficient awareness of computers. Managers should not consider computers as an evil but as an aid to their decision making. Similarly, they should ensure that the lower level users also consider it as a helpful tool in their day-to-day operations. Once this awareness is there, the management can press the users to define the correct requirements. After the requirements are given to the software developers, they should not be changed unless it is unavoidable. By unavoidable changes I mean those necessitated by virtue of new legislation or new technology etc Software recognition and education of all concerned can help in this regard as contemplated in the previous chapter.

C. USER ORIENTATION VS CUSTOMER ORIENTATION

By discounting the importance of user, it is by no means implied that software developers should develop what and how they feel like. Of course, they must give due consideration to the potential market and to the utility of the software

product, once it comes into the market. What is meant here is that a different stance should be taken. In the product concept, the software needs to be marketed, and as in marketing of any other product, customer is always the king. Therefore, his needs have to be met. It must be remembered that there is a difference between the earlier version of user satisfaction versus this customer satisfaction. The former denotes total dependency, whereas it is not so for the latter. The following points will further differentiate and clarify this issue.

1. User Orientation

- The developer is acquainted to the user to some extent or is organizationally related to the user.
- The user specifies his requirements directly to the developer.
- The user has one to one communication with the developer.
- The user participates in design reviews.
- The developer installs the software for the user.
- Problems in using the software are resolved by direct interaction between the user and the developer/ maintainer.

2. Customer Orientation

- The developer is neither acquainted with the users nor he has any organizational relationship with them.
- Requirements of the users are either deduced by the developer or are presented to him by an intermediary, such as a market research organization.
- There is no one to one communication between the user and the developer.
- Users do not participate in design reviews.

- Software is installed by the users themselves or someone else other than the developer does it for them.
- Problems are resolved through correspondence, sometimes through an intermediary.

VI. THE HUMAN FACTORS

Programming is a human activity and programmers are human beings. Elimination of the software crisis demands effective management, which in turn requires the managers to treat programmers as human beings and not as another machine constituting the computer configuration. Programming is a labor-intensive activity and human beings are the principal factor of software production. Therefore, to increase software productivity, the human factors warrant serious consideration by the managers.

Once we accept software developers as people first and then programmers, we have reason to believe that managerial theories and practices such as Herzberg's two factor theory of motivation, Maslow's hierarchy of needs, McGregor's theories X and Y, are all applicable to the software people to the same extent as they are to any other category of people.

A. MOTIVATION

Motivation is the means by which the potent wellsprings of human energy and creativity are directed towards people's desired goals. Most productivity studies have found that motivation is a stronger influence of productivity than any other contributing factor. [Ref. 24].

Motivation of software people is vital if we want to increase their productivity. First, we need to understand their objectives and then we need to incorporate these into the corporate decisions. We must also understand how the motivating factors of software people differ from other groups of people. A programming experiment was conducted by Weinberg which concluded that programmers have high

achievement motivation [Ref. 25]. If good achievement is defined with regards to what managers want form the project, the programmers will tend to work hard to give what is asked for. Another survey indicates that data processing people which consisted of a dominance of software people, are better motivated by growth needs than by social needs as shown in Figure 6.1 [Ref. 28]. One more study highlighted that distinction made by Herzberg between "hygienic factors" (supervision, administration, working conditions, salary, and inter-personal relations) and "motivating factors" (achievement, recognition, the work itself, advancement, self fulfillment, and participation), held good for data processing persons [Ref. 26]. However, marked differences were observed between the factor profiles of data processing persons and the overall population taken by Herzberg.

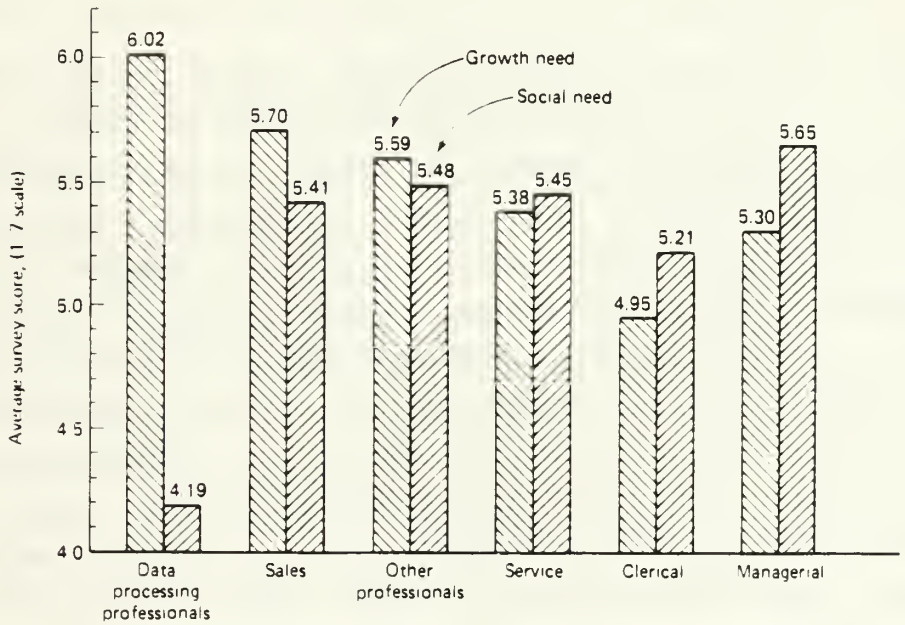

Figure 6.1 Comparative Growth Needs and Social Needs

TABLE 1

Comparison of First Level Motivational Factors
Data Processing Professionals Vs General Population

General Population [Herzberg and others 1959]	Data Processing Professionals [Fitz-Enz 1978]
1. Achievement	1. Achievement
2. Recognition	2. Possibility for growth
3. Work itself	3. Work itself
4. Responsibility	4. Recognition
5. Advancement	5. Advancement
6. Salary	6. Supervision, technical
7. Possibility for growth	7. Responsibility
8. Interpersonal relations, subordinate	8. Interpersonal relations, peers
9. Status	9. Interpersonal relations, subordinate
10. Interpersonal relations, superior	10. Salary
11. Interpersonal relations, peers	11. Personal life
12. Supervision, technical	12. Interpersonal relations, superior
13. Company policy and administration	13. Job security
14. Working conditions	14. Status
15. Personal life	15. Company policy and administration
16. Job security	16. Working conditions

The rank order of motivational factors in the survey of Herzberg and in Fitz-Enz survey are given in Table 1, which summarizes the differences in both. This summary also supports the results of Cougar-Zawacki [Ref. 27], which indicates high preference for growth amongst the data processing people. Growth was ranked seventh in Herzberg's survey, whereas it is second in Fitz-Enz survey. The other major differences were that data processing persons are more strongly motivated by opportunities for technical supervision, by peer relations, and by personal life. They are less strongly motivated by responsibility, salary, and status. Another interesting point is that these differences were more pronounced in case of software people than they were among other computer people.

Sometimes managers assume that lack of performance implies lack of motivation, and of course they are wrong.

Nonetheless, what they do is that they then try to supplement the lack of inner driving force with a little outer driving force, just when the programmer is suffering from too much. They should understand that increasing driving force will first increase performance to a maximum, beyond which addition of further driving forces will soon drive the performance to zero. It has been observed in research that this rapid fall-off is more pronounced in complex tasks, and programming is a considerably complex task. For instance, programmers are pressed hard to find out errors in their programs, quickly. Consequently, they try hard for the rapid elimination of the errors but they do not succeed.

In view of the findings of the surveys mentioned above, managers should keep in mind that the motivating factors for software people are different from those for other people. In the interest of having a better product, managers must give high priority to motivation of the software producers.

B. WATCH OUT FOR THE COMPULSIVE PROGRAMMER

In every computer center, one can find bright young men of disheveled appearance, often with sunken glowing eyes, sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to strike, at the keys on which their attention seems to be riveted as a gambler's on the rolling dice. If not in such a situation, they sit at tables full of computer printouts. They work for twenty to thirty hours at a stretch. They are not much concerned about food, and they sleep a few hours and then go back to the computer. They are not much concerned about their hygiene and bodies, and about the rest of the world. When they are involved in any job, they exist only for the computers. These are the kind known as compulsive programmers.

The compulsive programmers are distinguished from the professional ones, as the the latter address themselves to the problem to be solved, whereas the former see the problem merely as an opportunity to interact with the computer. The ordinary professional programmer usually discusses his programming problems with others. He does considerable planning before proceeding with the actual design and coding. He therefore, spends considerably less time on the computer, and may even allow others to key in his program. He is usually, organized and goes about doing his job systematically with a slow and steady pace. If he encounters some errors in the program, he will get away from the computer and look for the errors and bugs in a peaceful, non-computerized atmosphere. He will allow others to run his program, and thus the time which he saves, he spends on documenting the program and doing other beneficial works. He regards programming as a means to the end, not as an end itself. He gets satisfaction from solving a big problem, and not from bending the computer to his will.

The compulsive programmer is usually a good technician. He knows every detail of the computer he is working on. He is often tolerated in computer centers because of his knowledge of the system and because he can write small programs quickly, in one or two sessions of say, twenty hours, each. His programs are used in the computer center after some time, but there is a drawback to that. Since he can hardly be motivated to do anything except programming, his programs are not documented. Therefore, he is the only one who can understand his own programs. Consequently, he is assigned to teach his programs to others. He is like a bank employee who doesn't do much for the bank and yet he is retained because he knows combination to the safe. Usually, he likes to work on large programs. In making such programs, he has ambitious but imprecise goals. For example, he would like to create a

new computer language or create a system which can make it easier for others to write super-systems. He has the conviction that all such projects require nothing but computers and programming. Of course, he has lot of such knowledge, but during the process, when knowledge from outside the computer world is required, he is stuck.

The compulsive programmer spends almost all of his time, working on computer, but he doesn't call it working. Instead, he calls it "hacking." The dictionary meaning of "to hack" is "to cut irregularly, without skill or definite purpose; to mangle by or as if by repeated strokes of a cutting instrument." As mentioned earlier, he does have enough skills, but he is without definite purpose. He does not believe in setting forth a plan and goals, because he has the technique, no doubt, but he lacks knowledge.

Software systems can be built without plan and without knowledge, just as houses and buildings can be built in a similar manner. The important point here is that as the system becomes large, it also becomes unstable, when built in such a way. Eventually, it results in failure of the software and in extensive chaos. There is therefore, a word of caution for the managers. Managers! if you want to avoid software failures, set objectives and goals, plan and design properly, and have plenty of documentation. In other words, managers! watch out for the compulsive programmer. [Ref. 29].

C. STAFFING

Software productivity varies from individual to individual. This high degree of variation is ideally suited to enhance productivity by using the right people. The question of having the right mix of people leads us to the staffing principles which should be followed by the managers with

regards to the software persons. Boehm discusses five basic principles of software staffing, which are as follows [Ref. 28]. These principles are as follows.

1. The Principle of Top Talent
2. The Principle of Job Matching
3. The Principle of Career Progression
4. The Principle of Team Balance
5. The Principle of Phase Out

1. The Principle of Top Talent

Since there is a wide variation in productivity among different software practitioners, only the better people should be hired. Few better people will give a better output than many bad ones. Of course, they are going to cost more, but the additional cost will be offset by the benefits of having increased productivity.

2. The Principle of Job Matching

This principle suggests that the tasks should be fitted to the skills and motivation of the people. Managers should ensure that jobs are matched to the skills a person has, for instance, a person good in coding may not be good in documenting. A common violation of this principle arises when the programmers are promoted to management cadre. This usually does not work, and consequently much more mismatches, frustrations and damaged careers are observed in software engineering than in any other field. The reason for this is that on the average the data processing personnel have low social needs, whereas managers generally rank it high.

3. The Principle of Career Progression

The basic premise of this principle lies in helping the software people to self-actualize i.e. enabling them to

bring out the best in them. Software people achieve a good deal of self-actualization by becoming better software professionals. This principle highlights that that managers should help the software people determine how they want to grow professionally, and to provide them career development opportunities. Managers should curb the tendency amongst the software people to become irreplaceable. Instances do occur when a person does a job exceptionally well and thereafter he is always assigned that job. Such instances are frequently found in maintenance of software. Some software professionals become expert in maintaining certain piece of software, and so the managers do not allow them to work on anything else. Consequently, these people get stuck in this particular job, and eventually they feel better to quit the organization.

4. The Principle of Team Balance

This principle indicates that people should be selected who will complement and harmonize with each other. Apart from balancing the technical skills, the psychological factors peculiar to the software people, should also be balanced.

5. The Principle of Phase Out.

Survival of the fittest is what this principle stands for. Therefore, the software professionals who are not giving their optimum, should be eliminated. However,this should be the last resort. Before that, efforts should be made to rotate the person to some other job in which it is felt that he has interest. If all such efforts fail, then it is better to phase out the misfit person.

D. CAREER PATHS

As mentioned in the section on motivation, software professionals are motivated highly by growth, achievement and recognition. Therefore, managers must ensure that adequate career paths are established for them. Several ladders can be established, like

- Associate Software Engineer
- Assistant Software Engineer
- Software Engineer
- Senior Software Engineer
- Executive Software Engineer
- Chief Software Engineer

Alternatives for Engineer should also be established, such as "Evaluator" for the test function, "Writer" for the publications function, and "Analyst" for the support function. Formal descriptions should also be provided which define increasing responsibility and comparable experience for comparable titles. For instance if Senior Software Engineer requires ten years of experience, Senior Software Evaluator should also require the same amount of experience.

VII. CONCLUSIONS AND RECOMMENDATIONS

The preceding discussion somewhat understates the success of the software-producing sector of our society. Obviously, there have been substantial successes, which more than offset the negative effects. Scientists and technicians have done a lot and are still doing much. They have toiled hard, shedding enough of their sweat in developing a host of techniques and tools for software development. It's now time for the managers to step in and contribute their share. The situation is not so dismal as projected by most of the pundits in the software field--probably as a result of frustration.

In the not too distant future, our way of life is going to depend on the computer technology as it depends on technologies like electrical power, aircraft, automobile, radio and television. If there is a delay in the wide-spread application of computer technology, it will be because of software and not hardware. Managers must realize the existance of software crisis, and its serious impact on every walk of life in general, and to the business sector in particular. If optimum benefits are to be obtained from the new technology of computers, concerted efforts are required to eliminate the software crisis. Importance of software must be recognized and research efforts be put on the right path. For this purpose, software should be treated as an engineering discipline.

Omnipotent, panacean tools and techniques for software development do not exist. No software development tools or project management tools can compensate for the software engineer's lack of knowledge, skill and understanding. The

onus lies on the managers to ensure proper education of the software engineers, the users, and last but not the least, their ownselves. Curriculas for under-graduate and graduate education in software engineering need to be prepared, which should include an equal amount of managerial and technical courses. The curriculas proposed several years ago should be modified accordingly, and then implemented. A graduate course of study leading to the degree of Master of Science in Software Engineering (MSE), should be introduced widely. In-house training of experienced and qualified engineers in other disciplines can serve as a short-term solution.

To overcome the problem of guaranteeing the software product, the software engineers should be regarded as professionals. The Institute of Software Engineering (ISE), needs to be established, to carry out the tasks of certification of software engineers, and enforcement of code of ethics for them. Due to the acute shortage of software professionals, increasing their productivity warrants serious consideration. Enhancing their skills, and catering to the environmental and human factors can go a long way in obtaining optimum performance from the programmers.

Academicians and practitioners need to work more closely, and share their knowledge and experiences with each other. Similarly, managers and the technicians should learn the broad aspects of each others' disciplines. They should however, retain their existing jobs. In no event, the technical people should be made managers. They should have career paths in their own specialties. Sharing of knowledge extends to the programmers as well. Instead of "re-inventing the wheel," they should share each others' programs and learn lessons from each others' experiences.

Software should be treated as a product, similar to any other industrial product. It should be standardized, as much as possible. The organizations need to be encouraged to make use of the standard software product, unless there are other pressing requirements. In-house development and modifications of software be discouraged by the management. Software should be developed in-house only if there are peculiar needs of the organization. It should be modified only if there are inevitable changes. Requirements for software development should be ascertained from the market as a whole, and not from a few specific users, unless it is a customized product.

Managers must realize that software developers are human beings first and then programmers. Therefore, the personnel management theories and practices, are equally applicable to them as they are to any other category of people. They need to be motivated, and their peculiar needs are required to be fulfilled. They should be staffed properly, and career paths need to be established for them.

Shifting of software development tasks to the cheaper hardware provides the potentials of reducing the time and costs involved in the development process. A data bank of standard algorithms and modules needs to be established and economic evaluation of the technical tools and techniques is warranted.

If the fore-mentioned actions are taken, there is no doubt that technical and project management tools will yield fruitful results. The software industry in general, and the managers in particular, must strive to take the required actions, if there is a will to wipe out the software crisis, thereby obtaining the optimum advantages from the computer technology for the benefit of mankind.

LIST OF REFERENCES

1. Bukley, F.J. and Poston, R., "Software Quality Assurance," IEEE Transactions on Software Engineering, pp. 36-41, January 1984.

2. Mills, H.D., "Software Development," IEEE Transactions on Software Engineering, pp. 265-273, December 1976.

3. Heninger, K.L., "Specifying Software Requirements for Complex Systems: New Techniques and their Applications," IEEE Transactions on Software Engineering, pp. 2-12, January 1980.

4. Glass, R.L., Software Soliloquies, Computing Trends, Seattle, pp. 1-25, 1981.

5. Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw-Hill, Inc., New York, pp. 22-23, 1982.

6. Myers, G.J., The Art of Software Testing, John Wiley and Sons, New York, p. vii, 1979.

7. Baber, R.L., Software Reflected, North-Holland Publishing Co., New York, pp. 12-110, 1982.

8. Boehm, B.W., Software Engineering Economics, Prentice-Hall Inc., Englewood Cliffs, New Jersey, pp. 15-17, 1981.

9. Ibid, p.18.

10. Boehm, B.W., "Software and its Impact: A Quantitative Assessment," Datamation, pp. 48-59, May 1973.

11. Van Tassel, D.L. and Van Tassel, C.L., The Compleat Computer, Science Research Institute, Inc., Palo Alto, pp. 8-9, 1983.

12. Hoare, C.A.R., "Software Engineering: A Keynote Address," Proceedings of the 3rd International Conference on Software Engineering, pp. 1-2, 1978.

13. Wegner, P., Research Directions in Software Technology, MIT Press, Massachusetts, p.42, 1979.

14. Glass, R.L., Software Soliloquies, Computing Trends, Seattle, pp. 77-83, 1981.

15. Freeman, P., "A Proposed Curriculum for Software Engineering Education," Proceedings of the 3rd International Conference on Software Engineering, pp. 56-62, 1978.

16. McGill, J.P., "The Software Engineering Shortage: A Third Choice," IEEE Transactions on Software Engineering, pp. 42-49, January 1984.

17. Gunther, R.C., Management Methodology for Software Product Engineering, John Wiley and Sons, New York, pp. 2-16, 1978.

18. Wegner, P., Research Directions in Software Technology, MIT Press, Massachusetts, p.36, 1979.

19. Tou, J.T., Software Engineering, Academic Press, Inc., New York, pp. 116-136., 1970.

20. Comer, D., "Principles of Program Design Induced from Experience with Small Public Programs," IEEE Transactions on Software Engineering, pp. 169-173, March 1981.

21. Martin, J., Application Development Without Programmers, Prentice-Hall, Inc., New Jersey, 1982.

22. Liu, C., "A Look at Software Maintenance," Datamation, pp. 51-55, November 1976.

23. Scharer, L.L., "Pinpointing Requirements," Datamation, pp. 138-151, April 1981.

24. Gellerman, S.W., Motivation and Productivity, American Management Association Executive Books, New York, 1963.

25. Weinberg, G.M. and Schulman, E.L., "Goals and Performance in Computer Programming,"Human Factors, pp. 70-77, 1974.

26. Fitz-Enz, J., "Who is the DP Professional?" Datamation, pp. 124-129, September 1978.

27. Cougar, J.D. and Zawacki, R.A., "What motivates DP Professionals?" Datamation pp. 116-123, September 1978.

28. Boehm, B.W., Software Engineering Economics, Prentice-Hall Inc., Englewood Cliffs, New Jersey, pp. 667-675, 1981.

29. Van Tassel, D.L. and Van Tassel, C.L., The Compleat Computer, Science Research Institute, Inc., Palo Alto, pp. 61-63, 1983.

30. Sackman, H., Erikson, W.J. and Grant, E.E., "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," Communications of the ACM, January, 1968.

31. Schwartz, J., "Analyzing Large-Scale System Development," Proceedings of the 1968 Nato Conference, 1968.

32. Myers, G.J., " A Controlled Experiment in Program Testing and Code Walk-throughs/Inspections," Communications of the ACM, pp. 760-768, September 1978.

33. Rubey, R., "Higher Order Languages for Avionics Software - A Survey, Summary and Critique," NAECON, 1978.

BIBLIOGRAPHY

AFIPS-Time, A National Survey of the Public's attitudes Toward Computers, AFIPS and Time Inc., November 1971.

Alderfer, C. P., Existance, Relatedness and Growth: Human Needs in Organizational Settings, Free Press, New York, 1972.

Alford, M.W., "A Requirements Engineering Methodology for Real Time Processing Requirements," IEEE Transactions on Software Engineering, January 1977.

Arendt, H., The Human Condition, University of Chicago Press, Chicago, 1958.

Aron, J.D., The Program Development Process: The Individual Programmer," Addison-Wesley, Massachusetts, 1974.

Basili, V.R. and Reitter, R.W., "An Investigation of Human Factors in Software Development," Computer, December 1979.

Basili V., Models and Metrics for Software Management and Engineering, IEEE Computer Society Press, Los Angeles, 1980.

Bass, L.J. and Oldehoeft, R.R., "Dynamic software science with applications," IEEE Transactions on Software Engineering, September 1979.

Bendix, R., Work and Authority in Industry, John Wiley and Sons, New York, 1956.

Biggs, C., Managing the System Development Process, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

Boehm, B.W., "Software Engineering," IEEE Transactions on Computers, December 1976.

Bratman, H. and Court, T.,"The Software Factory," IEEE Computer, May 1975.

Brooks, F.P., The Mythical Man-Month, Addison-Wesley,Massachusetts,1975.

Bryan, W. and Siegel, S., "Making software visible operational and maintainable in a small project environment," IEEE Transactions on Software Engineering, January 1984.

Chen, E.T.,"Program Complexity and Programmer Productivity," IEEE Transactions on Software Engineering, May 1978.

Cofer, C. and Appley, M., Motivation: Theory and Research, John Wiley and Sons, New York, 1964.

Conway, M.E., "On the Economics of the Software Market," Datamation, October 1968.

Conway, M.E., "How do Committees Invent?" Datamation, April 1968.

Crosby, P.B., Quality Is Free: The Art of Making Quality Certain, McGraw-Hill, New York, 1979.

Daly, E.B., "Managing Software Engineering," IEEE Transactions on Software Engineering, May 1977.

Daly, E.B., "Organizing for Successful Software Development," Datamation, December 1979.

Davis, K., Human Relations at Work, McGraw-Hill, New York, 1962.

DeMarco, T., Controlling Software Projects, Yourdon Press,New York,1982.

DeMarco, T., Report on the 1977 Productivity Survey, Yourdon Press, September 1977.

Donaldson, H., A Guide to the Successful Management of Computer Projects, Wiley, New York, 1978.

Elbing, A.O., Behavioral Decisions in Organizations, Scott Foresman and Co., Illinois, 1978.

Elsalmi, A.M. and Cummings, L.L., "Managerial Motivation: The Impact of Role Diversity, Job Level, and Organizational Size," Proceedings of the Academy of Management, 1968.

Esterling, B., "Software Manpower Costs: A Model," Datamation, March 1980.

Fairley, R.E.,"Software Engineering Education: Status and Prospects," Proceedings of the Twelfth Hawaii International Conference on System Sciences, Western Periodicals Ltd., California, 1979.

Fordyce, J.K. and Weil, R., Managing With People, Addison-Wesley,Massachusetts,1971.

Frank, W.L., "The New Software Economics: Parts 1-4," Computerworld, January 1979.

Garrity, J., Getting the Most out of Your Computer, Mckinsey, New York, 1963.

Gildersleeve, T.R., Data Processing Project Management, Von Nostrand Reinhold, New York, 1971.

Glass, R.L., "Persistent Software Errors," IEEE Transactions on Software Engineering, March 1981.

Gordon, R.L. and Lamb, J.C., " A Close Look at Brooks' Law," Datamation, June 1977.

Hansson, R.O. and Fiedler, F. E., "Perceived Similarity, Personality and Attraction to Large Organizations," Journal of Applied Psychology, vol. 3, no. 3, 1973.

Hoare, C.A.R., "The Emperor's Old Clothes," Communications of the ACM, February 1981.

Homans, G.C., The Human Group, Harcourt, Brace and World, New York, 1950.

Jones, C., Programming Productivity : Issues for the Eighties, IEEE Computer Society Press, Los Angeles,1981.

Jones, C.B., Software Development: A Rigourous Approach, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

Kanter, J., Management-Oriented Management Information Systems, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

Kerr, S., Organizational Behavior, Grid Publishing Inc., Ohio, 1979.

Kindred, A., Data Systems and Management, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

Koontz, H. and O'Donnel, C., Principles of Management: An Analysis of Managerial Functions, McGraw-Hill, New york, 1972.

Kraft, P., Programmers and Managers: The Routinization of Computer Programming in the United States, Springer-Verlag, New York, 1977.

Lamb, C.A.,"DP and the user: A matter of planning," Datamation, November 1978.

Lientz, B. and Swanson, E., Software Maintenance Management, Addison-Wesley, Massachusetts, 1980.

Levinson, H., Psychological Man, Levinson Institute, Cambridge, Massachusetts, 1976.

Martin, J., Software for Distributed Processing, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

Maslow, A.H., Motivation and Personality, Harper and Row, New york, 1954.

Maslow, A.H.,"A Theory of Human Motivation," Psychological Review, vol. 50 (1943).

Maslow, A.H., Euypsychian Management, Irwin Dorsey Press, Homewood, 1965.

McCracken, D.D., "Software in the 80s: Perils and Promises," Computerworld, September 17, 1980.

McGregor, D., The Human Side of Enterprise, McGraw-Hill, New York,1960.

Merwin, R.E.,(ed),"Special Section on Software Management," IEEE Transactions on Software Engineering, July 1978.

Metzger, P.W., Managing a Programming Project, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

Monash, P., "Software Strategies," Datamation, February 1984.

Nelson, E.A., Management Handbook for the Estimation of Computer Programming Costs, Systems Development Corp., October 31,1966.

Newell, A. and Simon, H., Human Problem Solving, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

Patrick, R.L., "Probing Productivity," Datamation, September 1980.

Phister, M.Jr., Data Processing Technology and Economics, Santa Monica Publishing Co. and Digital Press, California, 1979.

Porter, L.W., Behavior in Organizations, McGraw-Hill, New York, 1975.

Reifer, D., Software Management, IEEE Computer Society Press, Los Angeles, 1979.

Richards, M.D., and Nielander, W.A., Readings in Management, South-Western Publishing, Cincinnati, 1962.

Ridge, W.J. and Johnson, L.E., Effective Management of Computer Software, Dow-Jones Irwin, Illinois, 1973.

Roche, W.J. and Mackinnon, N.L., "Motivating People with Meaningful work", Harvard Business Review, May-June 1970.

Ross, D.T. and Brackett, J.W., "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, January 1977.

Sayles, L.R., The Behavior of Industrial Work Groups, Wiley, New York, 1958.

Schneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Winthrop Press, Cambridge, Massachusetts, 1980.

Scott, R.F. and Simmons, D.B., "Programmer Productivity and the Delphi Technique," Datamation, May 1974.

Sharpe, W.F., The Economics of Computers, Columbia University Press, New York, 1969.

Shaw, J.C. and Atkins, W., Managing Computer System Projects, McGraw-Hill, New York, 1970.

Sutermeister, R.A., People and Productivity, McGraw-Hill, New York, 1963.

Tajima, D. and Matsubara, T., "The Computer Software Industry in Japan," Computer, May 1981.

Thurber, K., Computer System Requirements, IEEE Computer Society Press, Los Angeles, 1980.

Uris, A., Mastery of Management, Dow Jones-Irwin Inc., Illinois, 1968.

Viteles, M., Motivation and Morale in Industry, Norton, New York, 1953.

Webber, R.A., Management: Basic Elements of Managing Organizations, Richard D.Irwin Inc., Illinois, 1979.

Weinberg, G.M., "The Psychology of Improved Programming Performance," Datamation, November 1972.

Weinwurm, G.F., On the Management of Computer Programming, Auerbach, New York, 1970.

Weizenbaum, J., Computer Power and Human Reason, W.H.Freeman, San Francisco, 1976.

Whyte, W.F., Man and Organization, Richard D. Irwin Inc., Homewood, Illinois, 1959.

Wiener, N., The Human Use of Human Beings: Cybernetics and Society," Doubleday and Co., New York, 1954.

Wilkes, M.V., "Software engineering and structured programming," IEEE Transactions on Software Engineering, December 1976.

Witt, J., "The COLUMBUS Approach," IEEE Transactions on Software Engineering, December 1975.

Yourdon, E., Managing the System Life Cycle: A Software Development Methodology Overview, Yourdon Press, New York, 1982.

Yourdon, E., Classics in Software Engineering, Yourdon Press, New York, 1979.

Zaleznik, A. and Moment, D., Interpersonal Dynamics, Wiley, New York, 1964.

Zunde, P., "Empirical Laws and Theories of Information and Software Sciences," Information Processing and Management, August 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey California 93943	2
3.	Computer Technology Programs, Code 37 Naval Postgraduate School Monterey California 93943	1
4.	Associate Professor Clair A. Peterson Code 54Pe Department of Administrative Sciences Naval Postgraduate School Monterey California 93943	1
5.	Associate Professor Norman R. Lyons Code 541b Department of Administrative Sciences Naval Postgraduate School Monterey California 93943	1
6.	Lieutenant Commander Tahir N. Qureshi 5/42 Model Colony Karachi 27, Pakistan	2

Thesis
Q842 Qureshi
c.1 Attacking software crisis a macro approach.